Application Note 28

The ARM7TDMI Debug Architecture

Document Number: ARM DAI 0028A

Issued: December 1995

ARM
Advanced RISC Machines

Change Log

Issue	Date	By	Change
A	Dec 95	BP/AP/EH	Created

THE ARM7TDMI Debug Architecture

Table of Contents

THE ARM7TDMI Debug Architecture

1 Introduction

A vital stage of any product development cycle is the debugging and testing of the system. With the increasing complexity of designs, the software development and system debugging stage of a product now contributes to a significant proportion of the time to market and in order to remain competitive the product development cycle needs to be kept to a minimum. In deeply embedded designs the microprocessor core is not directly accessible from the periphery of the chip, adding to the problem of debugging the system.

This application note describes how the problem has been overcome by the ARM7TDMI Debug Architecture and the advantages of using this approach.

2 The ARM Debug Architecture—an Overview

The ARM Debug Architecture uses a protocol converter box to allow the debugger to talk via a JTAG (Joint Test Action Group) port directly to the core. In effect the scan chains in the core that are required for test are re-used for debugging.

The architecture uses the scan chains to insert instructions directly in to the ARM core. The instructions are executed on the core and depending on the type of instruction that has been inserted, the core or the system state can be examined, saved or changed. The architecture has the ability to execute instructions at a slow debug speed or to execute instructions at system speed (for example if access to an external memory was required).

The fact that the debugger is actually using the JTAG scan chains to access the core is of no importance to the user as the front end debugger remains exactly the same. The user could still use the debugger with a monitor program running on the target system or with an instruction set simulator that runs on the debugger host. In each case the debugging environment is the same.

The advantages of using the JTAG port are:

- hardware access required by a system for test is re-used for debug.
- core state and system state can be examined via the JTAG port.
- the target system does not have to be running in order to start debug.

  A monitor program for example requires that some target resources are running in order for the monitor program to run.
- the traditional breakpoints and watchpoints are available.
- on-chip resources can be added to.

  For example the ARM Debug Architecture uses and an on-chip macrocell to enhance the debugging facilities available.
- a separate UART to communicate with the monitor program is not required.

The debugging of the target system requires the following:

- a host computer to run the debugger software. The host could be a PC running Windows, a Sun workstation or an HP workstation.
- an Embedded ICE Protocols Converter. A separate box which converts the serial interface to signals compatible with the JTAG interface and a target system with a JTAG interface and an ARM Debug Architecture compliant core.

Figure 1: ARM debug system on page 4 shows how the system is connected.

Once the system is connected, the debugger can start communicating with the target system via the Embedded ICE Interface Converter.

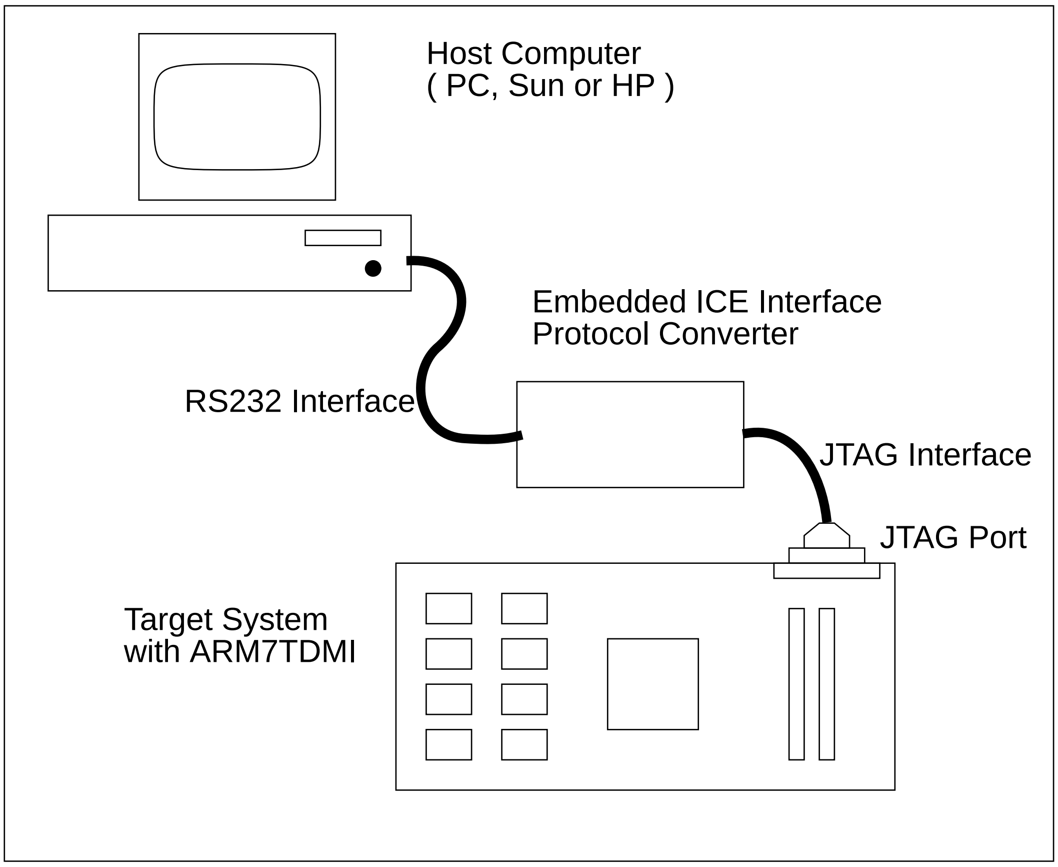


Figure 1: ARM debug system

3 The ARM7TDMI Debug Architecture

The ARM7TDMI Debug Architecture uses the existing JTAG (Joint Test Action Group) port as a method of accessing the core. The scan chains that are around the core for production test are reused in debug state to capture information from the databus and to insert new information into the core or the memory.

There are effectively two scan chains around the core:

- a scan chain around the whole periphery of the core
- a subset of the first scan chain, covering only the databus and breakpoint

The shorter scan chain on the databus allows instructions and data to be inserted into the core without the overhead of clocking the data around the entire periphery of the ARM7TDMI processor core.

In addition to the scan chains, the ARM7TDMI Debug Architecture uses a macrocell called the EmbeddedICE macrocell. The EmbeddedICE macrocell resides on chip with the ARM7T processor core. The EmbeddedICE has its own scan chain that is used to insert watchpoints and breakpoints for the ARM7TDMI processor core.

The ARM Debug Architecture allows programs to execute at full system speed and for the ARM7TDMI processor core to be halted by the EmbeddedICE macrocell when a breakpoint or watchpoint is seen by the EmbeddedICE macrocell. The ARM7TDMI processor core then enters a debug state which allows the internal system state to be examined and changed if required. The system can be restarted once the debug session is over.

The ARM7TDMI processor core has three additional signals that are controlled by the EmbeddedICE macrocell to force the processor into and out of Debug state. The signals are:

DBGRQ	Debug Request. This is a level sensitive input which, when HIGH, causes the ARM7TDMI processor core to enter debug state after executing the current instruction. This allows external hardware to force the ARM7TDMI processor core into the debug state.
DBGACK	Debug Acknowledge. This signal is an output from the ARM7TDMI processor core which, when HIGH, indicates that the ARM7TDMI processor core is in debug state.
BREAKPT	Breakpoint. This signal is an input to the ARM7TDMI processor core. When HIGH, the current memory access is breakpointed. If the memory access is an instruction fetch, the ARM7TDMI processor core will enter debug state if the instruction reaches the execute stage of the ARM7TDMI processor core pipeline. If the memory access is for data, the ARM7TDMI processor core will enter debug state after the current instruction completes execution.

The debug signals can also be brought to the periphery of the ASIC so that external hardware can control the ARM7TDMI processor core if required.

4 Use of JTAG Scan Cells

The JTAG specification (IEEE 1149.1) was originally developed to aid the testability of larger ASIC devices. The technique uses scan cells that can apply inputs to the macrocell while isolating the input from the rest of the circuit and can also sample outputs from the macrocell.

The inputs and outputs can be applied and sampled serially through a JTAG port. The JTAG port is the connection to the rest of the circuit. The JTAG port connections are:

TMS	Test Mode Select. The **TMS** signal selects the next state in the TAP state machine.
TCK	Test Clock. This allows shifting of the data in, on the **TMS** and **TDI** pins. It is a positive edge-triggered clock with the **TMS** and **TCK** pins that define the internal state of the device. A sixteen state finite state machine, with certain states allocated functions within the JTAG specification, is controlled by the clocks and the test mode.
TDI	Test Data In. This is the serial data input for the shift register. It connects to all the storage elements in the device and allows shifting of binary data patterns to either set up the inputs to the macrocell or the outputs to the system internal scan chain.
TDO	Test Data Output. This is the serial data output from the shift register. Data is shifted out of the device on the negative edge of the **TCK** pin. The data out is either the results from the macrocell or the signals being applied to it.
nTRST	Test Reset. The test reset pin is an optional pin. The **nTRST** pin can be used to reset the test logic within the macrocell. The pin is optional because it is possible to force the test logic into reset using a combination of the **TMS** and **TCK** pins.

The configuration of the scan cell is shown in ***Figure 2: Scan cells***.

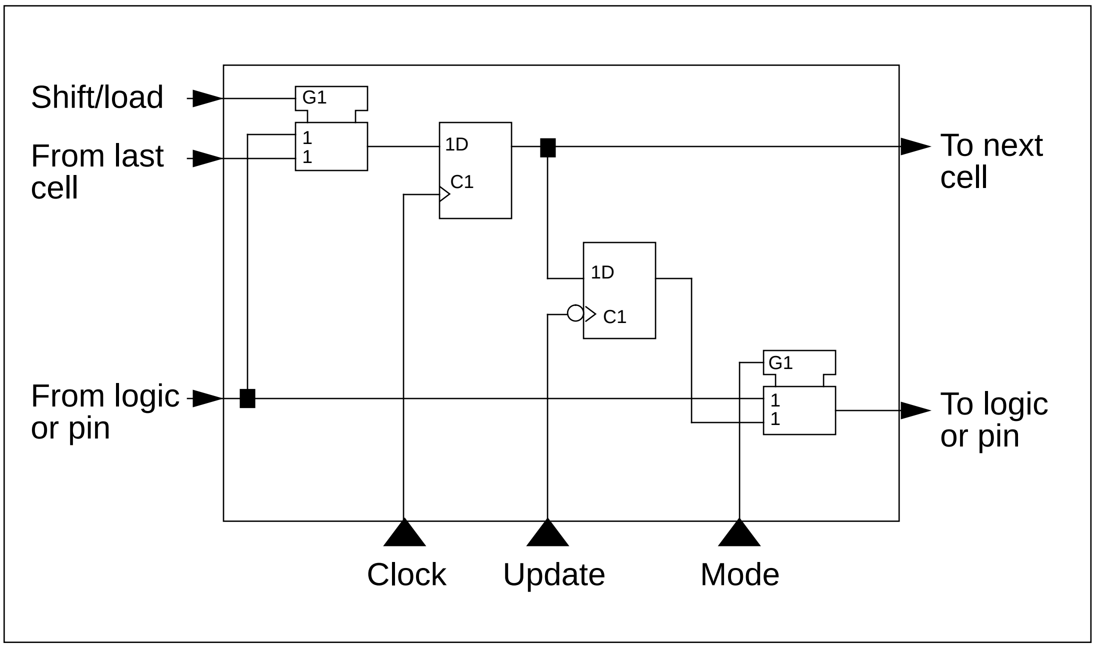


Figure 2: Scan cells

The scan cell is placed on an input or an output of the ARM macrocell. The scan cells can be configured to:

- apply inputs to the macrocell
- sample outputs from the macrocell
- link the scan cells together in a shift register that allows the data to be shifted in or out of the macrocell via the **TDI** and **TDO** pins

5 Use of the Scan Chains

Figure 3: Example device with scan cells below shows a small example of a circuit with scan cells around it. For clarity the control signals to each scan cell have been omitted.

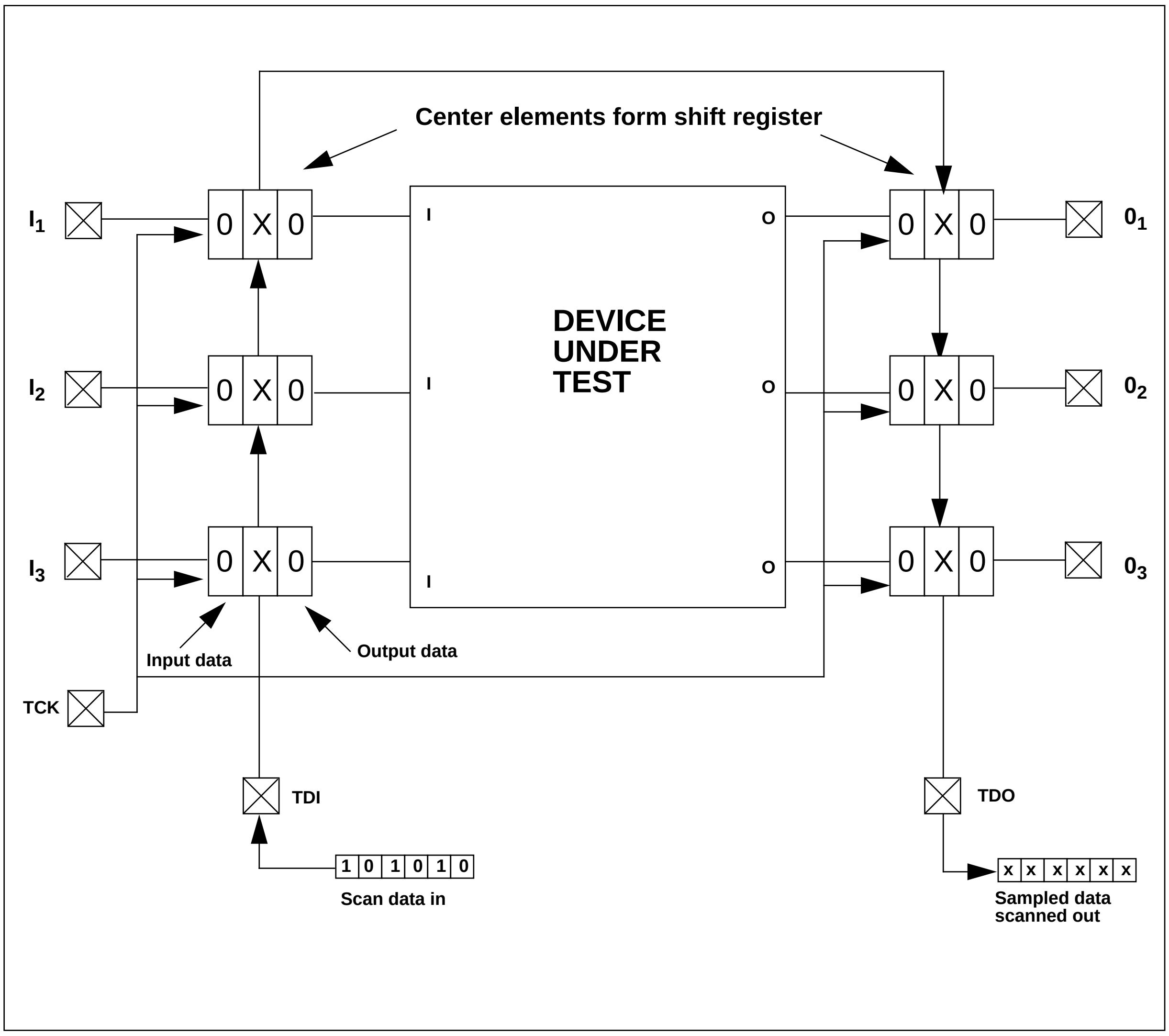


Figure 3: Example device with scan cells

Figure 4: Sequence of diagrams on page 9 shows how the outputs from the macrocell can be sampled and scanned out of the device and new inputs to the macrocell scanned in and applied to the device.

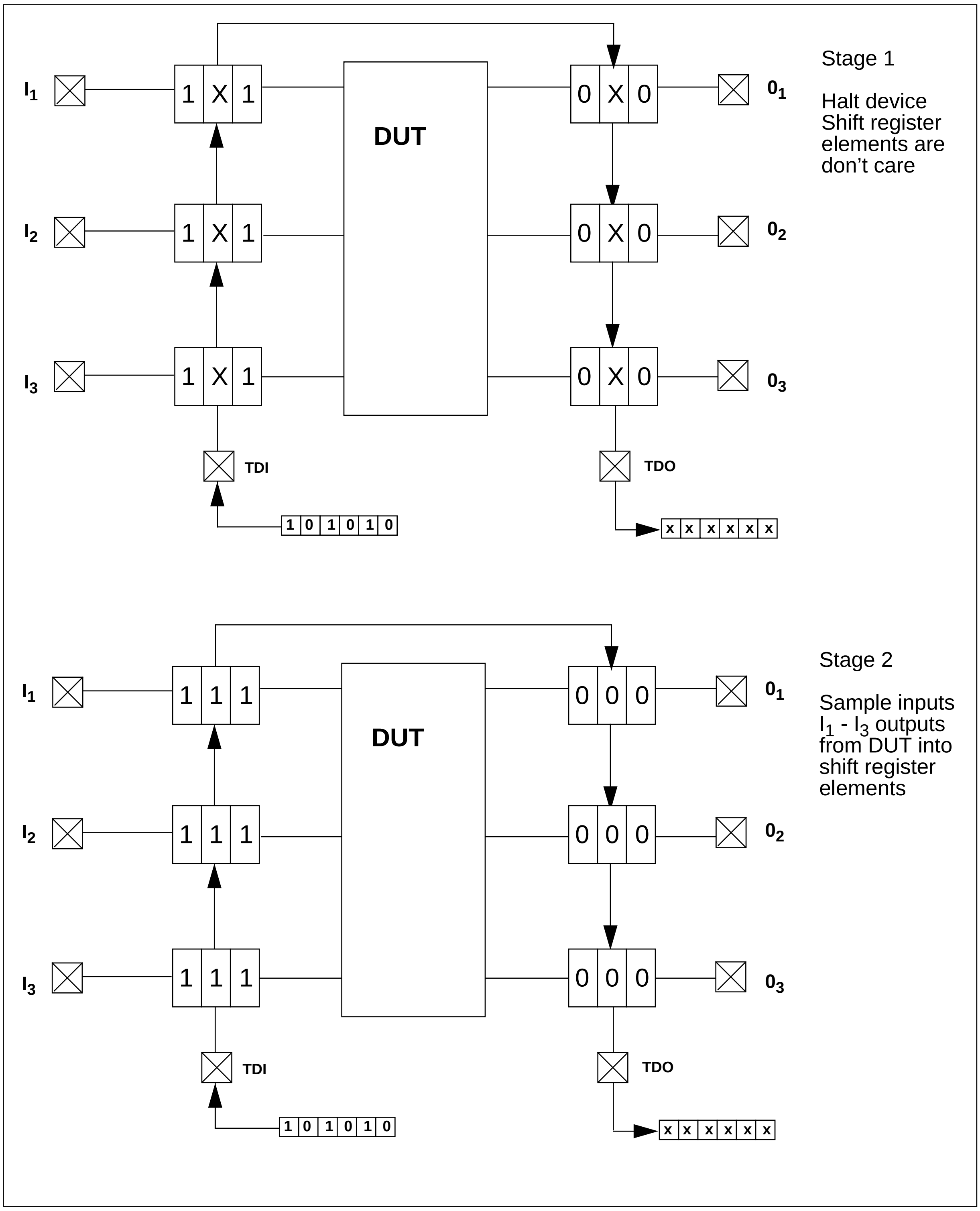
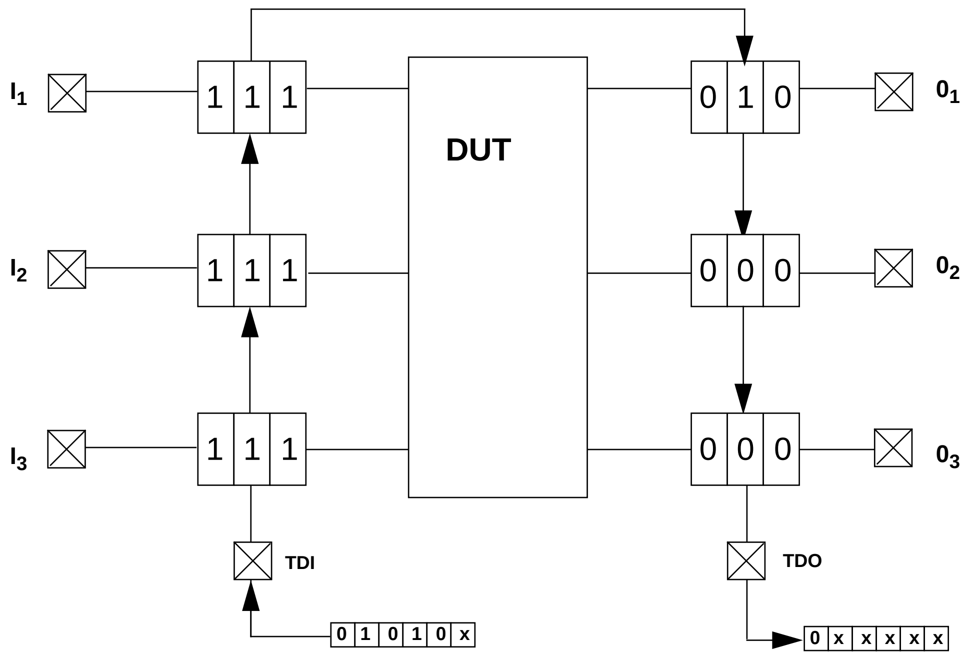
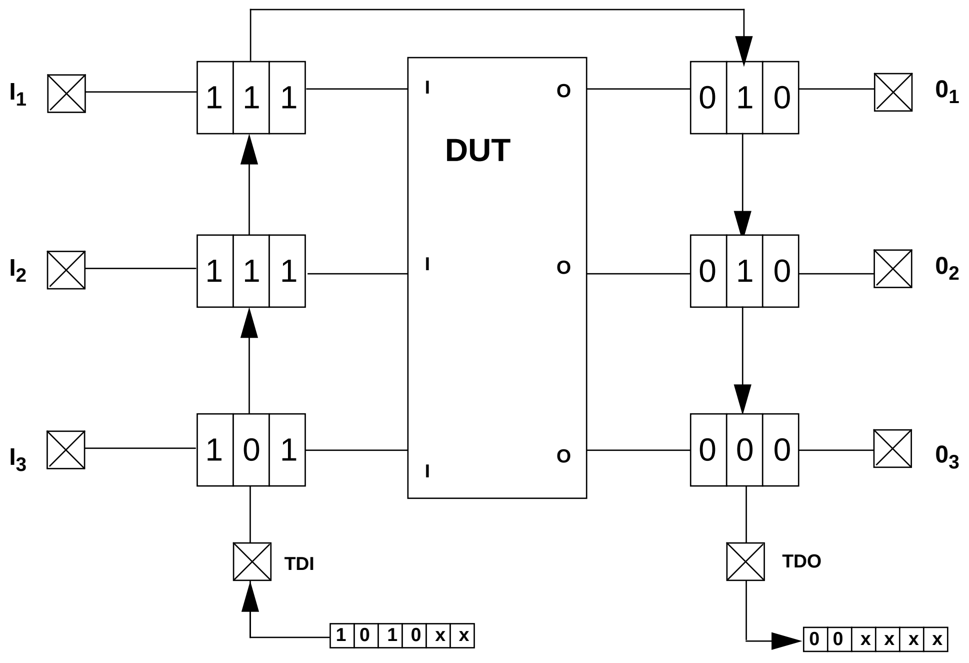

Stage 1
Halt device Shift register elements are don't care
DUT
I1
I2
I3
O1
O2
O3
1 X 1
0 X 0
TDI
TDO
1 0 1 0 1 0
x x x x x x
Stage 2
Sample inputs I1 - I3 outputs from DUT into shift register elements
1 1 1
0 0 0

Figure 4: Sequence of diagrams



Stage 3

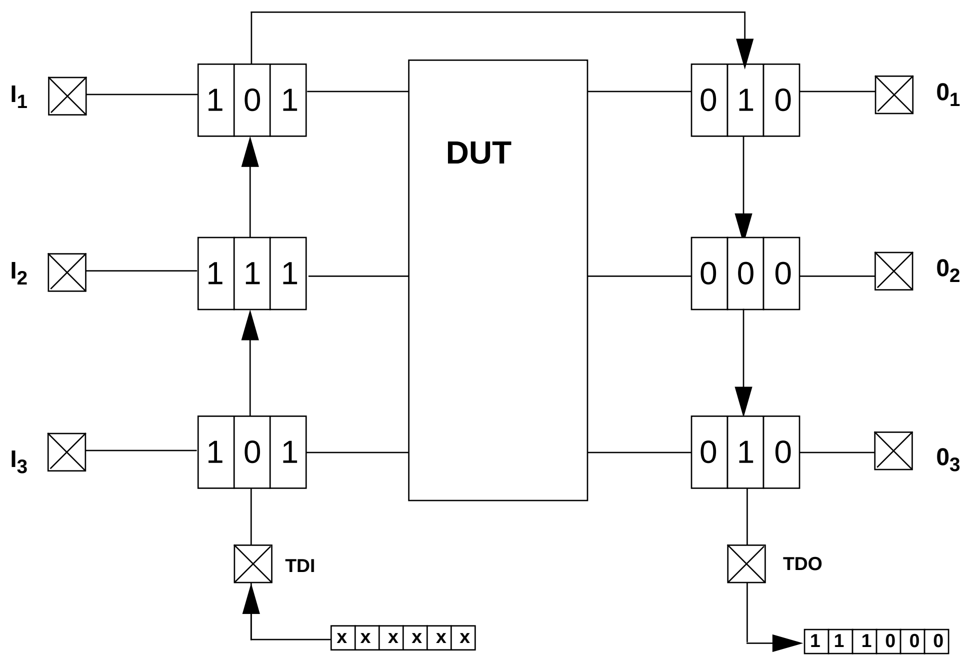
Clock **TCK** to move shift register along one place. First sample clocked into I_3 scan cell from input buffer. The sample on O_3 scan cell clocked into output buffer.



Stage 4

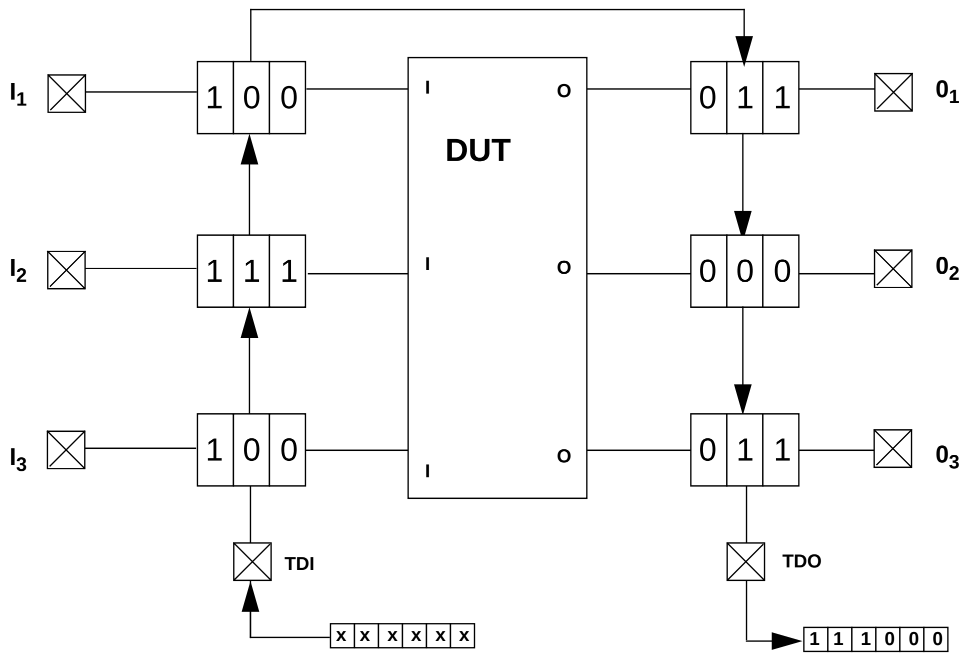
Second tick moves shift register along another place. Sample from O_2 is now in the output buffer and the input buffer has two values moved out.

Figure 4: Sequence of diagrams (continued)



Stage 5

After 6 TCKs the value originally in the input buffer is now in the scan chain. The input and outputs originally sampled by the scan chain are now in the output buffer.



Stage 6

New inputs now applied to the DUT and new outputs applied from the DUT. The DUT can now be run normally until new inputs are required to be applied or new outputs sampled.

Figure 4: Sequence of diagrams (continued)

Data can be sampled from the outputs of the macrocell and applied to the inputs of the macrocell through a five wire interface.

If an input or output, to or from the macrocell is required to be unchanged, the value scanned into the scan chain will be the same as was previously present on the input or output.

6 Configuration of the Scan Chains in the ARM macrocell

Figure 5: Scan chains in the ARM core shows the configuration of the scan chains in the ARM macrocell. The scan chains are arranged as follows:

Scan Chain 0	around the entire periphery of the macrocell. All the inputs and outputs can be controlled via scan chain 0.
Scan Chain 1	around the databus and breakpoint only. Scan chain 1 is used to scan instructions and data into the macrocell. The scan chain has been shortened to reduce the time it takes to insert instructions into the macrocell. It avoids the penalty of scanning data around the periphery of the macrocell just to control the 32 bits of the databus.
Scan Chain2	around the EmbeddedICE macrocell. The scan chain is used to control the registers in the EmbeddedICE macrocell.

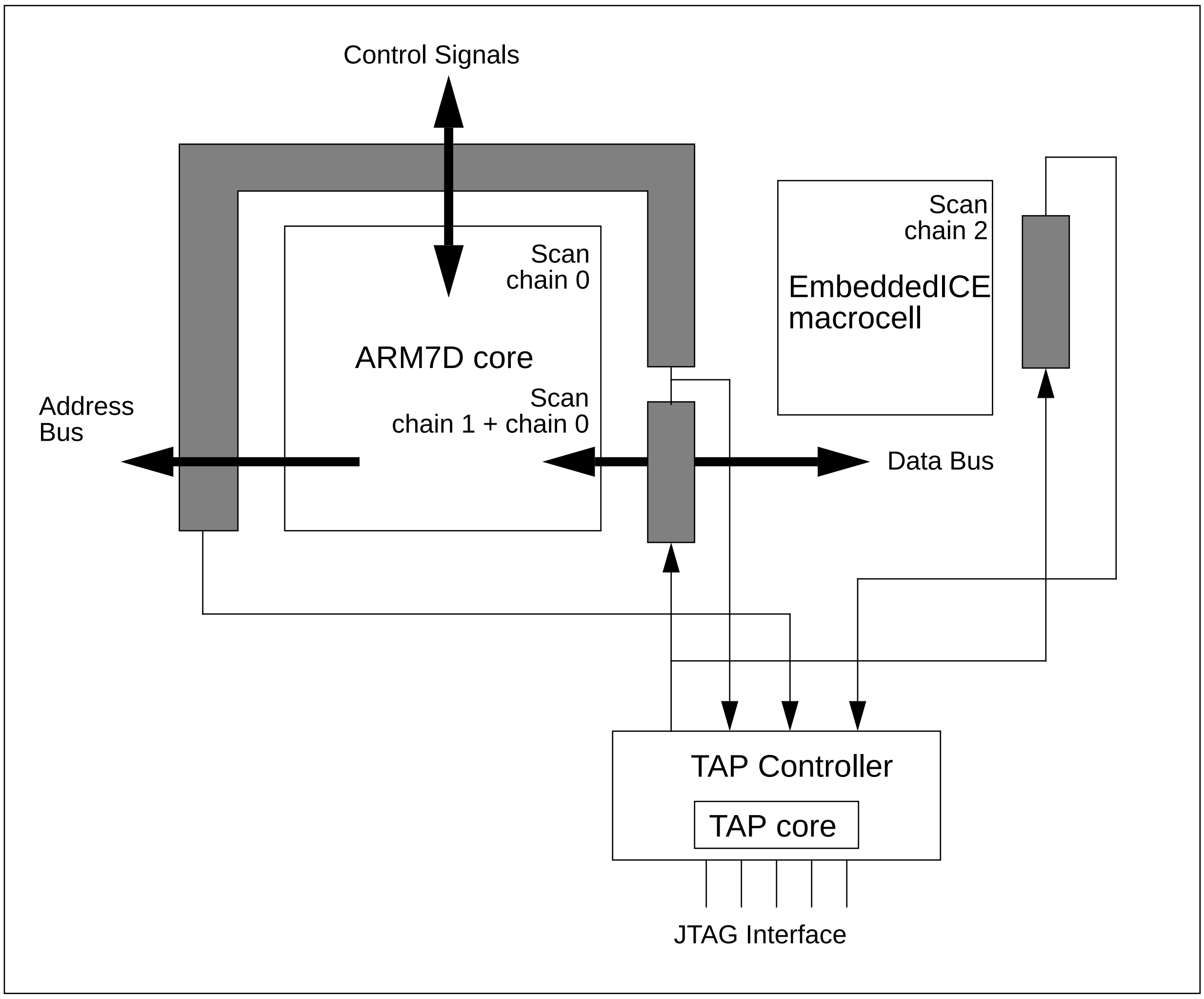


Figure 5: Scan chains in the ARM core

THE ARM7TDMI Debug Architecture

The ARM Debug architecture allows certain instructions to be performed at system speed. For example a load multiple instruction (LDM) can be used at system speed to examine the contents of some memory locations quickly. To do this, the ARM7TDMI core needs to know which instructions are to be performed at system speed (running at the **MCLK** frequency) and those which run at debug speed (running at **DCLK** frequency). This is done by adding an extra scan cell to the scan chain across the databus. When an instruction is scanned into the databus scan chain, the extra scan cell is set to identify to the ARM7TDMI core that the next instruction should be executed at system speed. When the instruction reaches the execute stage of the pipeline, the ARM7TDMI core must resynchronize back to **MCLK** in order to execute the instruction. Once the instruction has completed, the core will synchronize back to **DCLK**.

7 The EmbeddedICE Macrocell

The EmbeddedICE macrocell provides on-chip debug support for the ARM7TDMI core. The EmbeddedICE macrocell consists of two real time watchpoint registers, together with a control and status register. One or both of the watchpoint registers can be programmed to halt execution of instructions by the ARM7TDMI core via the **BREAKPT** signal. Execution is halted when a match occurs between the values programmed into the EmbeddedICE macrocell and the values currently appearing on the address bus, databus and some control signals. Any bit can be masked so that its value does not affect the comparison.

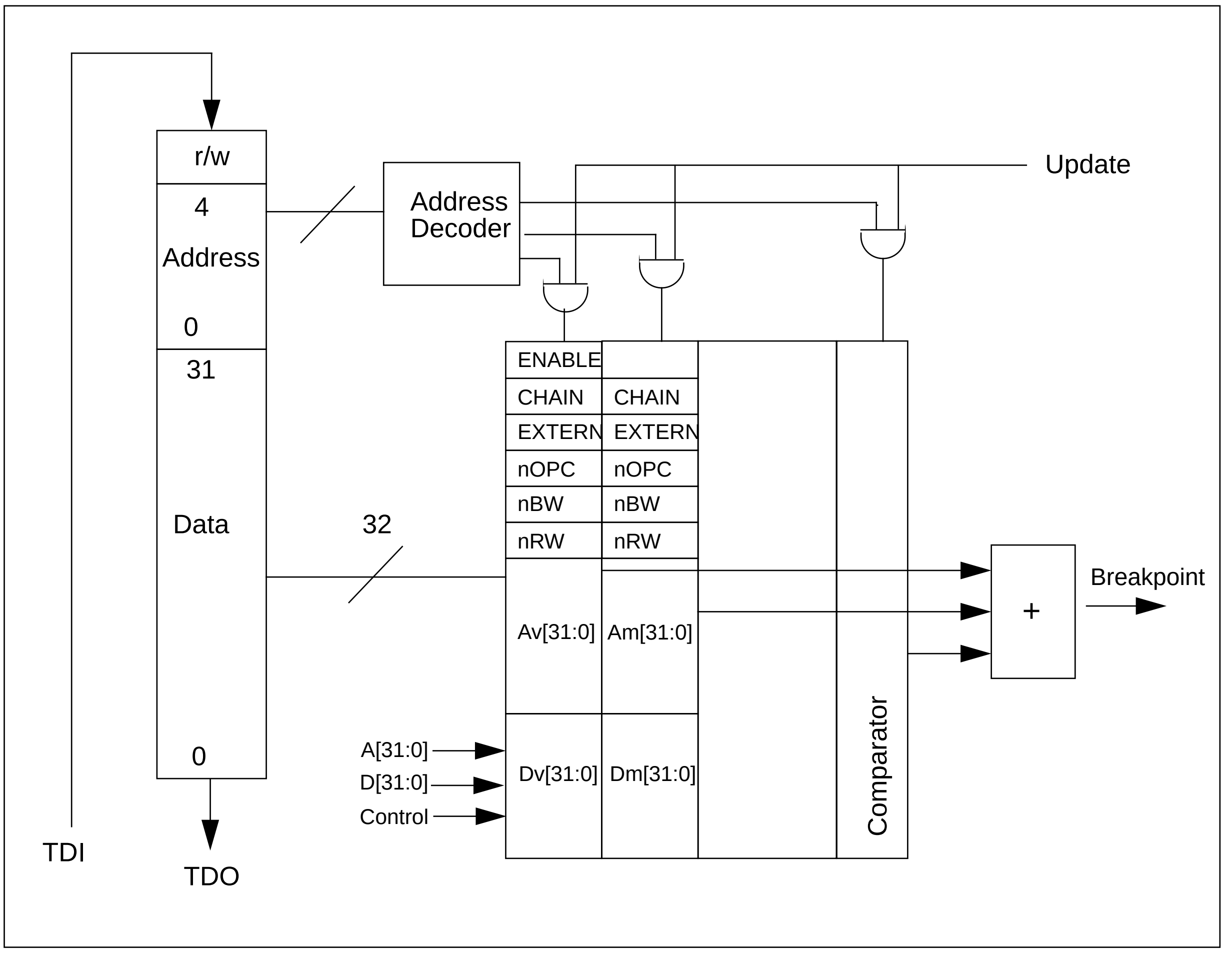


Figure 6: EmbeddedICE macrocell

Either watchpoint can be configured as a watchpoint (ie. on a data access) or a breakpoint (ie. on an instruction fetch).

The watchpoints and breakpoints can also be combined such that:

- the conditions on both watchpoints must be satisfied before the ARM7TDMI core is stopped. The CHAIN functionality allows two consecutive conditions to be satisfied before the core is halted. An example of this would be to set the first breakpoint to trigger on an access to a peripheral and the second to trigger on the code segment that performs the task switching. Therefore when the breakpoints trigger the information regarding which task has switched out will be ready for examination.
- the watchpoints can be configured such that a range of addresses are enabled for the watchpoints to be active. The RANGE function allows the breakpoints to be combined such that a breakpoint is to occur if an access occurs in the bottom 256 bytes of memory but not in the bottom 32 bytes.

The ability to CHAIN or RANGE the breakpoints considerably enhances their usefulness.

In addition to the CHAIN and RANGE functionality the watchpoints can also be enabled or disabled by using two external inputs to the ARM7TDMI core. The inputs are called **EXTERN0** and **EXTERN1**.

EXTERN0	enables or disables watchpoint 0
EXTERN1	enables or disables watchpoint 1

An external input can also be used to control if the watchpoint is enabled or disabled. An example might be that debugging is only required during a particular sequence in the product functionality and at all other times the ability to stop the core and enter debug state is disabled.

8 Debug Signals

The signals that are external to the core and used in debug state were described in ***3 The ARM7TDMI Debug Architecture*** on page 5 but the names and functionality are recapped here.

DBGRQ	Debug Request. This is a level sensitive input, which when HIGH causes the ARM7TDMI processor core to enter debug state after executing the current instruction. This allows external hardware to force the ARM7TDMI processor core into the debug state.
DBGACK	Debug Acknowledge. This signal is an output from the ARM7TDMI processor core which when HIGH indicates that the ARM7TDMI processor core is in debug state.
BREAKPT	Breakpoint. This signal is an input to the ARM7TDMI processor core. When HIGH the current memory access is breakpointed. If the memory access is an instruction fetch the ARM7TDMI processor core will enter debug state if the instruction reaches the execute stage of the ARM7TDMI processor core pipeline. If the memory access is for data, the ARM7TDMI processor core will enter debug state after the current instruction completes execution.

In addition to these three signals there is a fourth signal that can be used to enable or disable the debug functionality on the ARM7TDMI core. The reason for having a signal to do this is for safety critical applications where it is essential that the developer can guarantee that the ARM7TDMI core will not be able to enter debug state. The signal is called **DBGEN** and is described below.

DBGEN	Debug Enable. This signal allows the debug features of the ARM7TDMI to be disabled completely. The signal should be driven LOW when debugging is not required.

The timing associated with the debug signals is shown in ***Figure 7: Debug timing*** and ***Figure 8: Breakpoint timing***.

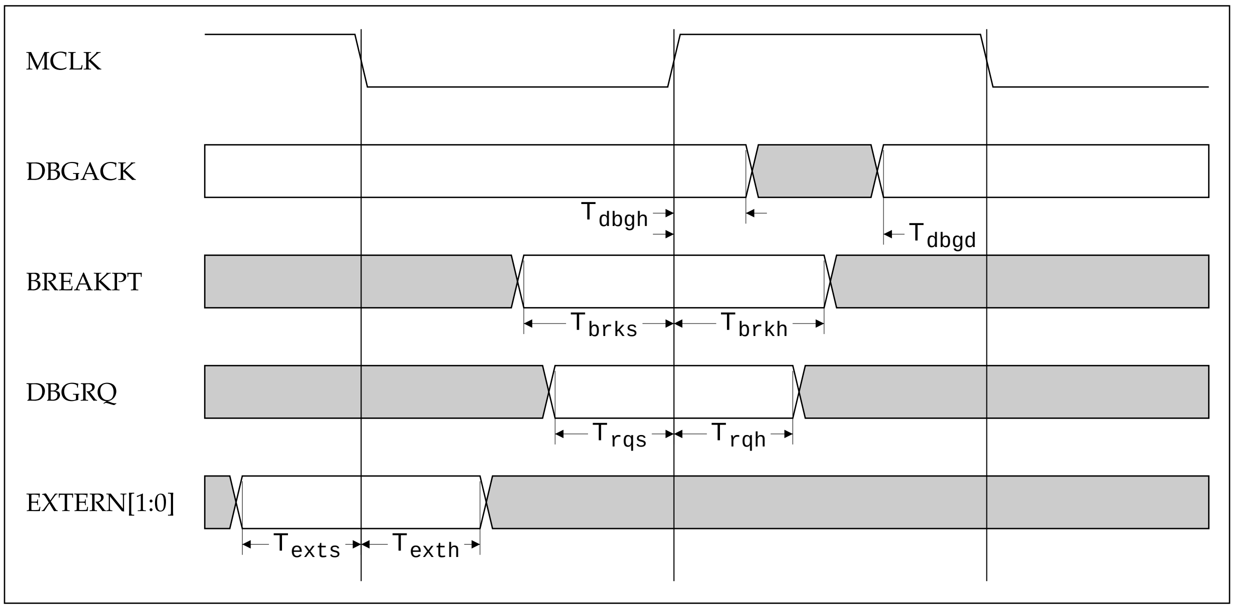


Figure 7: Debug timing

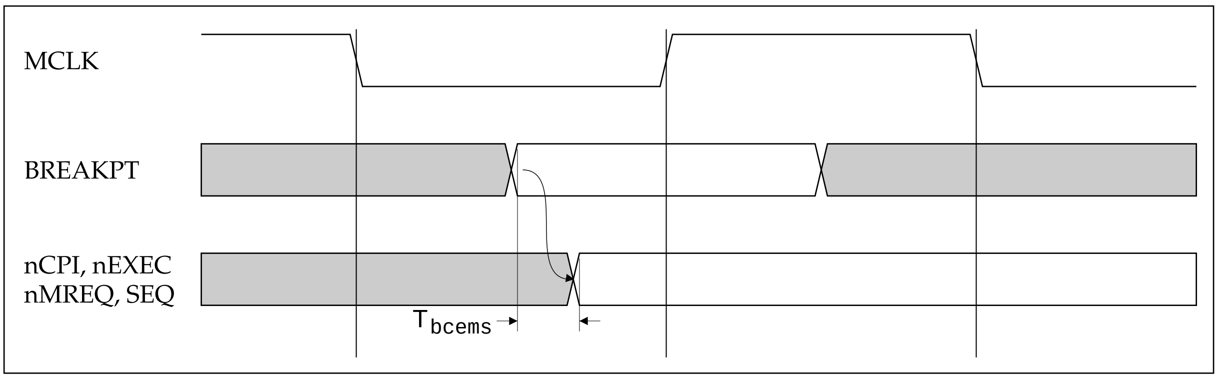


Figure 8: Breakpoint timing

Figure 7: Debug timing and ***Figure 8: Breakpoint timing*** show that the main debug signals (**DBGACK**, **BREAKPT** and **DBRQ**) are timed relative to the rising edge of **MCLK**. The **EXTERN** signals are sampled on the falling edge of **MCLK**.

Once the ARM7TDMI core has entered debug state the ARM7TDMI core appears to be in an idle state relinquishing the databus such that any external activity can occur in real time. The core may still be accessed via the JTAG port, allowing the host computer to insert instructions and data.

9 Entering Debug State

The ARM7TDMI core will enter debug state under any one of the following conditions:

- when a breakpointed instruction reaches the execute stage of the instruction pipeline
- a memory access occurs on data that has a watchpoint on it
- the **DBGRQ** signal is forced HIGH

Once the breakpoint has occurred the ARM7TDMI must switch from using **MCLK** to using a debug clock for the core called **DCLK**. **DCLK** is controlled from the TAP state machine. When entering debug state this is handled automatically by the core. On entry to debug state the core asserts **DBGACK** to signal to the external system that the core is now in debug state.

For simplicity it will be assumed that the debug state is entered from ARM state.

10 Determining the Core State

The instruction used to examine the processors internal state is:

```
STM   R0, { R0 - R15 }
```

This causes the contents of the registers to be made visible on the databus. The sequence of events is as follows:

1. **DCLK** from the TAP state machine clocks the contents of the first register out onto the databus.
2. The databus value and hence the contents of the first register are captured into scan chain 1.
3. The contents of scan chain 1 are clocked out of the JTAG port via **TDO** using **TCK**. The contents of the first register are now available to the debugger.
4. **DCLK** now clocks the contents of the second register out onto the databus.
5. The databus value and hence the contents of the second register are captured into scan chain 1.
6. The contents of scan chain 1 are clocked out of the JTAG port via **TDO** using **TCK**. The contents of the second register are now available to the debugger.
7. The process is repeated until all sixteen registers have been clocked out of the core, captured in scan chain 1 and clocked out to the debugger via scan chain 1.

Figure 9: Clock and data control shows how the clocks and data are controlled.

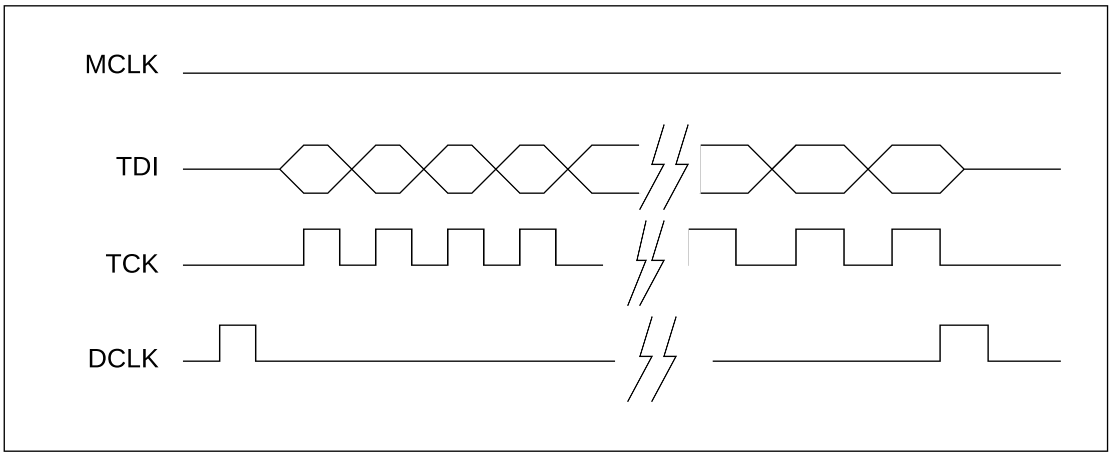


Figure 9: Clock and data control

The STM instruction is said to execute at debug speed. Debug speed is much slower than system speed since between each core clock, 33 scan clocks occur in order to shift in an instruction or shift out data. Executing the instructions more slowly than is usual does not cause a problem as the ARM7TDMI core is fully static.

While in debug state only the following instructions may legally be scanned into the instruction pipeline for execution:

- all data processing instructions, except **TEQP**
- all load, store, load multiple and store multiple instructions
- **MSR** and **MRS**

11 Determining System State

The system state cannot be examined as easily as the core state due to the dynamic timing requirements of the other components in the system. DRAMs, for example, can only be accessed at system speed, otherwise they will not work correctly and data cannot be read from or stored to the memory.

To overcome this problem, any attempt to examine the system state must take place at normal system speed. The ARM7TDMI core can be synchronized to run from **MCLK** under the control of the bit 33 in scan chain 1. If bit 33 of the scan chain is set to a 1, the next instruction will be executed at system speed. The ARM7TDMI core needs to be resynchronized back to system speed before the instruction that will actually be executed at system speed is in the execute stage of the pipeline. ***Table 1: Use of bit 33 to control system speed access*** shows this.

Instruction	Bit 33	Comments
`STM R0, {R0-R15}`	0	Examine core registers
`MOV R0, R0`	0	NOP
`MOV R0, R0`	1	NOP but LDM instruction to be executed at system speed
`LDM R0, {R0-R15}`	0	Load multiple executed at system speed

Table 1: Use of bit 33 to control system speed access

After the system speed instruction has been scanned into the databus and clocked into the pipeline, the BYPASS instruction must be loaded into the TAP controller. This will cause the ARM7TDMI to automatically synchronize back to **MCLK** (the system clock), execute the instruction at system speed, re-enter debug state and switch itself back to an internally generated **DCLK**.

The debugger will load the TAP controller with the INTEST instruction to allow the debugging session to resume.

The use of system speed load multiples and debug speed store multiples allow the core and system memory state to be fed back to the host.

12 Exiting Debug State

The internal state of the ARM7TDMI core must be restored before exiting debug state. Debug state is exited by executing a branch to the next instruction to be executed at system speed.The penultimate instruction to be scanned into the ARM7TDMI core must have bit 33 of scan chain 1 set to a 1 to cause the last instruction of the debug sequence, the branch, to be executed at system speed.

The ARM7TDMI core is actually restarted at system speed by loading the RESTART instruction into the TAP controller. When the state machine enters the RUN TEST/IDLE state, the scan chain will revert back to system mode and the clock synchronization back to **MCLK** will occur within the ARM7TDMI. The ARM7TDMI will then resume normal operation fetching instructions from the destination of the branch.

13 Entering Debug from Thumb State

The ARM7TDMI could enter debug state from ARM state or THUMB state. The first thing an external debugger must do is to find out if the ARM7TDMI was in ARM or THUMB state before entering debug state. This can be achieved by examining bit 4 of the EmbeddedICE macrocell’s Debug Status register. To do this, the external debugger controls the JTAG port in the following sequence:

1. Scan the SCAN_2 instruction into the TAP controller to select scan chain 2 in the EmbeddedICE macrocell.
2. Scan the address of the Debug Status register into scan chain 2 with the read-write bit set low in order to read the register.
3. Scan the data out of scan chain 2.
4. Examine bit 4 of the Debug Status register that has been scanned out serially.

If the processor was in Thumb state, the simplest course of action is to force the core back into ARM state. To force the ARM7TDMI processor into ARM state the following sequence of Thumb instructions should be executed on the core:

```
STR   R0, [R0] ; Save R0 before use
MOV   R0, PC   ; Copy PC into R0
STR   R0, [R0] ; Now save the PC in R0
BX    PC           ; Jump into ARM state
MOV   R8, R8   ; NOP
MOV   R8, R8   ; NOP
```

Note *Since all Thumb instructions are only 16 bits long, the simplest course of action when shifting them into scan chain 1 is to repeat the instruction twice. The debugger does not then have to keep a track of which half of the databus the ARM7TDMI core is expecting to read from.*

14 Control Registers within the EmbeddedICE macrocell

The EmbeddedICE macrocell contains a total of 16 registers. The registers are a combination of control, data, status and mask registers. ***Table 2: EmbeddedICE macrocell registers*** shows the registers contained in the EmbeddedICE macrocell.

Address	Width	Function	Comments
00000	3	Debug Control	Force debug state, disable ints
00001	5	Debug Status	Status of debug, TBIT
00100	6	Debug Comms Control Register	
00101	32	Debug Comms Data Register	
01000	32	Watchpoint 0 Address Value	
01001	32	Watchpoint 0 Address Mask	
01010	32	Watchpoint 0 Data Value	
01011	32	Watchpoint 0 Data Mask	
01100	9	Watchpoint 0 Control Value	
01101	8	Watchpoint 0 Control Mask	
10000	32	Watchpoint 1 Address Value	
10001	32	Watchpoint 1 Address Mask	
10010	32	Watchpoint 1 Data Value	
10011	32	Watchpoint 1 Data Mask	
10100	9	Watchpoint 1 Control Value	
10101	8	Watchpoint 1 Control Mask	

Table 2: EmbeddedICE macrocell registers

Note *The Debug Communication Channel is described in* ***15 The ARM7TDMI Debug Communication Channel*** on page 24*.*

The registers in the EmbeddedICE macrocell for the watchpoints each have a data register and a mask register. For example the address for watchpoint 0 could be programmed to a particular value and the data value either ignored completely by setting the mask value or the data value can be used to set the condition for the watchpoint. The use of the data mask value means, for example, that only if the value from that address is a certain value will it cause a watchpoint.

The registers are repeated for the watchpoint 1.

The watchpoint control registers allow the watchpoint to also be dependent an some of the external signals. For example **EXTERN[0:1]** can be enabled from these registers.

15 The ARM7TDMI Debug Communication Channel

The ARM7TDMI has an extra function built into it called the *debug communication channel*. The debug communication channel is a method by which a program running on the ARM7TDMI core can communicate with the host debugger or another separate host without stopping the program flow or even entering debug state. The debug communication channel allows the JTAG port to be used for sending and receiving data without affecting the normal program flow.

The advantage of this channel is that the debug communication channel can be used as a simple method of communicating with the ARM7TDMI core from the outside world or to monitor the program flow in the ARM7TDMI core by sending monitoring messages from the ARM7TDMI core as different tasks within the program are entered.

The ARM7TDMI debug communications channel is accessed as a coprocessor by the program running on the ARM7TDMI core. The coprocessor is number 14. By writing to and reading from coprocessor 14, the program can communicate with the host. Two registers are used by the debug communications channel. The debug communication channel register is a read-only register and allows for synchronized handshaking between the processor and the debugger. Two bits in the control register can be read to determine if the debug communications write register is free to write to, and whether there is new data in the debug communications read register.

The instructions used by the program to access the debug communications channel registers are:

`MRC CP14, 0, Rd, C0, CO`	returns the Debug Comms Control Register into Rd
`MCR CP14, 0, Rn, C1, C0`	writes the value in Rn to the Comms Write Register
`MRC CP14, 0, Rd. C1, C0`	returns the Debug data read register into Rd

Note *Since the THUMB instruction set does not contain coprocessor instructions these will need to be written as SWI (Software Interrupt) Handlers when in THUMB state. Entering the SWI handler immediately puts the ARM7TDMI into ARM state where the coprocessor instructions are available.*

The debugger cannot use coprocessor 14 to access the debug communications channel as this has no meaning to the debugger. The debugger can read and write to the debug communications channel registers using the scan chain. The debug communication channel data and control registers are mapped into addresses in the EmbeddedICE macrocell.

The sequence of events for the ARM7TDMI core to communicate with the debugger are as follows:

1. The processor checks that the debug communication channel write register is free for use. It does this using the MRC instruction to read the debug communications channel control register to check that the W bit is clear.
2. If the W bit is clear, the debug communication write register is clear and the processor can write a message to it using the MCR instruction to coprocessor 14. The action of writing to the register automatically sets the W bit.

If the W bit is set, the debug communication write register has not been emptied by the debugger. The processor must then poll the W bit until it is clear.

3 The debugger polls the debug communications control register via scan chain 2. If the debugger sees that the W bit is set it can read the debug communications channel data register to read the message sent by the processor. The process of reading the data automatically clears the W bit in the debug communication control register.

Message transfer from the debugger to the processor is the reverse process.

1 The debugger polls the debug communication control register R bit. If the R bit is clear, the debug communication read register is clear and data can be written there for the processor to read.

2 The debugger scans the data into the debug communication read register via scan chain 2. The R bit in the debug communication control register is automatically set by this.

3 The processor polls the R bit in the debug communication control register. If the R bit is set, there is data in the debug communication read register that can be read by the processor using the MRC instruction to read from coprocessor 14.

The processor accesses the debug communications registers as a coprocessor register transfer read and write whilst the debugger accesses the same registers via the scan chain.

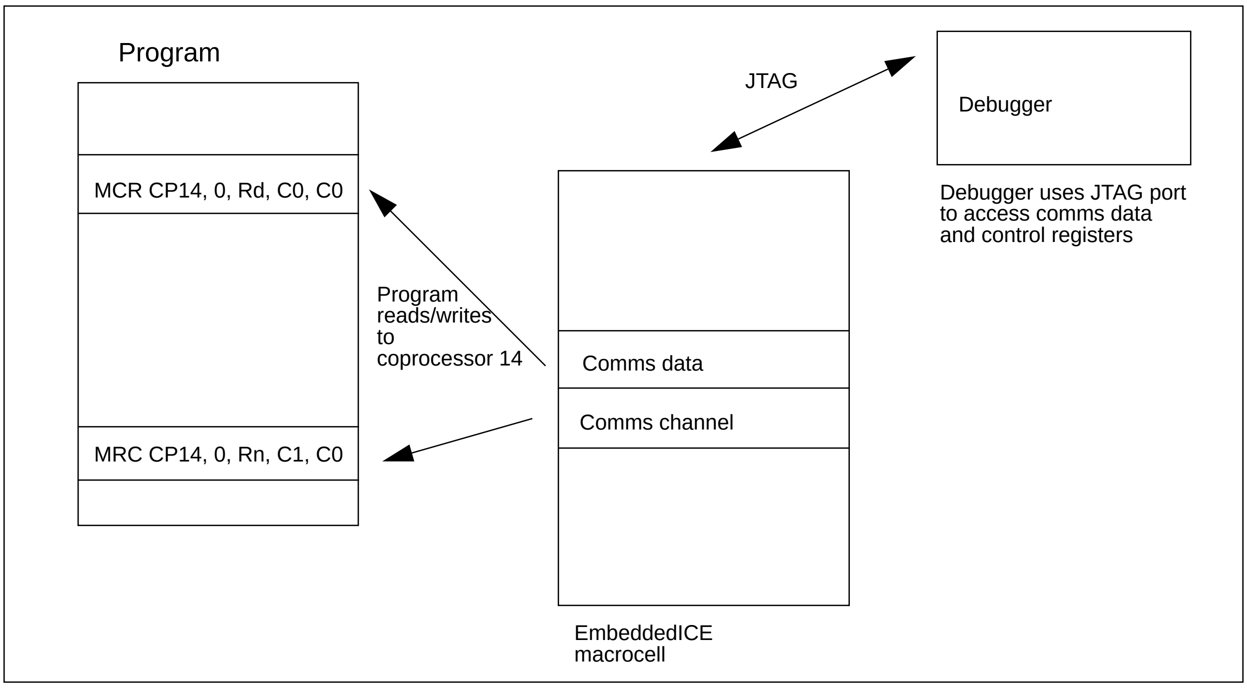


Figure 10: Debug communications channel

Figure 10: Debug communications channel shows how the debug communications channel works.

16 Implications for ARM Debuggers

The ARM Software Development Toolkit debuggers do not need to know that the target is an ARM7TDMI with an EmbeddedICE macrocell. The debuggers still talk to the parallel or serial port on the host using the ARM Remote Debug Protocol. The system uses a converter box called the EmbeddedICE Interface Protocol converter that takes the Remote Debug Protocol commands and converts them into the JTAG data needed to access the ARM7TDMI. The user will not notice any difference from using a conventional debug monitor running on the target system.

The big advantage of using the ARM Debug Architecture is that no target resources are required by the debugger in order to start the debugging session.